JOURNAL OF COMPUTER AND SYSTEM SCIENCES 35, 23–58 (1987)

Communication in Concurrent Dynamic Logic

DAVID PELEG

IBM Almaden Research Center, 650 Harry Road, San Jose, California 95120

Received March 1, 1985

Communication mechanisms are introduced into the program schemes of Concurrent Dynamic Logic, on both the propositional and the first-order levels. The effects of these mechanisms (particularly, channels, shared variables, and "message collectors") on issues of expressiveness and decidability are investigated. In general, we find that both respects are dominated by the extent to which the capabilities of synchronization and (unbounded) counting are enabled in the communication scheme.

PART A: INTRODUCTION AND PRELIMINARIES

1. *Introduction*

One of the most widely studied models of concurrent computation is that of an *and/or tree.* This approach views concurrency in its purest form as the *dual* notion of nondeterminism. For example, let us illustrate a process by a tree whose arcs represent "atomic actions." Nondeterminism is introduced by allowing a node to split into several branches, and letting the process choose between the different possible continuations. Analogously, concurrency is obtained by again splitting a node into several branches, but requiring the process to execute *all* possible continuations independently. This is basically the classical concept of and/or decomposition, which occurs widely in computability, logic, game theory, etc.

Addressing the problem of appropriate programming tools for concurrency, the described model gave raise to the *concurrent goto programs* appearing in the works of Manna and Chandra [Ma, Ch]. These programs may contain such commands as *goto* l_1 *or* l_2, facilitating nondeterministic choice, as well as commands like *goto* l_1 *and* l_2, causing a split into two parallel independent branches.

This approach to parallel computation was also introduced by Chandra *et al.* [CKS] into the theory of computability, through the mechanism of *Alternating Turing Machines* (ATMs); the computation tree of an ATM can be represented by an and/or tree. ATMs have proved to be a very useful tool in complexity theory, and have been followed by several attempts to develop programming tools incorporating and implementing this duality, like Harel's *And/Or Programs* [H4], the

0022-0000/87 $3.00

language *Ind* of [HK] and, in a way, the new trend of logic programming and *Prolog* [K, ClM, S2].

Concurrent Dynamic Logic (CDL) is introduced in [Pe1] as an extension of regular Dynamic Logic ([FL], cf. [H1]), which attempts to provide a framework for reasoning about concurrent programs in the and/or model. CDL is discussed on both the propositional and the first-order levels, and is shown to possess most of the desirable properties of regular DL, while being capable of modeling and discussing concurrency in a satisfactory manner.

However, parallel processes in CDL are isolated, mutually oblivious, and unsynchronized. Not only can they not communicate with one aother, but it is impossible even to collect their "final reports" by some main process, for a global analysis. This is, of course, quite a serious limitation; in actual concurrent programming languages, much of the power is based on the ability of individual processes to communicate with one another. This is the case in *CSP* [Ho], *CCS* [M], *Concurrent And/Or programs* [HN] and *Concurrent Prolog* [S1], not to mention a large variety of distributed systems, in which communication plays a vital role.

All this has motivated the present research, in which we examine some extensions to CDL, tailored towards incorporating communication mechanisms into the programs. The resulting logics are eventually intended to serve as a basic ground for reasoning about concurrent programs with communication, and thus are to play the same role as the one played by DL with respect to regular (flowchart) programs.

Let us first characterize the roles and tasks required from communication in a concurrent environment. The main functions of communication can be roughly categorized as

(1) synchronization of processes,

(2) exchange of information between processes, and

(3) transmission of reports, messages and final results to some main "parent" process.

In this paper we investigate several versions of communication schemes for CDL. These versions vary by the mechanism used, such as channels vs. common memory, and by the extent to which communication is allowed. We consider both the propositional and the first-order levels of CDL. The propositional versions were meant to provide an abstraction of the discussed notions, constructs, and mechanisms, so as to enable an analysis of their fundamental properties and behaviour.

In general, our results indicate that the scope of communication crucially affects the issues of expressiveness and decidability. In particular, allowing full propositional communication between processes (by means of either channels or shared variables) renders the logic highly undecidable. In order to get a decidable propositional logic it is necessary to limit the features of synchronization (hidden in

the "blocking" semantics of channel communication) and unbounded counting (hidden in the ability to spawn an unbounded number of processes); either one of these causes undecidability. We present such a decidable prototype system.

We now give a brief description of the logics considered in this paper. Communication on the propositional level is based on Boolean (two valued) messages. We begin with **channel**-CPDL (here P stands for propositional). This language enables communication between two parallel processes by means of a channel. A process may transmit the message 0/1 along channel C (by executing the command $C!0/C!1$, respectively), or receive 0/1 (by using $C?0/C?1$). The notation is taken from Hoare and Milner [Ho, Mi].

These operations cannot be carried out independently by a single process; rather, it takes a pair of processes wiling to transmit and receive the same message simultaneously. Actually, a channel C is merely a programming construct, representing no actual entity in the model. Thus the simultaneous execution of, say, $C!1$ and $C?1$ in two parallel processes may be viewed as the verification of two tests, or "guards," by the processes. The fact that **channel**-CPDL allows full communication, including complete synchronization of processes and unbounded counting, renders the validity problem of **channel**-CPDL undecidable (in fact, Π^1_1-hard), as is shown using methods of [H3].

Defining a communication scheme based on shared variables leads to similar behaviour. We discuss such a system named **shared**-CPDL, and its connections with previous extensions proposed for PDL (**shared**-CPDL requires us to extend the semantic foundations of the system to include additional auxiliary variables, cf. [A], or to allow a program to change the truth value of propositional variables, cf. [T]). We show that **shared**-CPDL is equivalent to **channel**-CPDL, under certain conditions which are necessary in order to make them compatible.

The third and last mechanism considered is based on *counts*. This mechanism restricts communication considerably, and allows only the transmission of messages M_i from the processes to a main control process. This control is again only an abstract entity, represented by its semantic functioning.

There are several viable ways of defining the functions of such a main process. In the systems proposed here these functions are quite limited. The main control process can recognize different messages and count the number of messages arrived of each type (either precisely or modulo some constant), but it cannot identify the order in which the messages arrived or the source of each message (unless the number of processes is kept bounded in the program, in which case it is possible to identify each process by giving it a special message-set). We actually consider two versions of this system. The more powerful one is **u-count**-CPDL. This version is actually equivalent to a weakened version of **channel**-CPDL, with a non-blocking semantics for communication (i.e., a version in which a process may send a message even when no other process waits to receive it, using an intermediate infinite buffer). Thus the synchronization capability is eliminated. Still, **u-count**-CPDL allows a comparison of the number of messages sent of two types, and the unbounded counting hidden in this capability renders this version Π^1_1-hard too. A more limited

version is **count**-CPDL, which does not provide this facility, and therefore is strictly less expressive than **u-count**-CPDL, and moreover, is elementarily decidable.

On the first-order level, communication is based on general messages, i.e., the contents of a message is some element of the domain.

The main versions considered are **channel**-CQDL and **shared**-CQDL (here Q stands for quantified), which are the first-order counterparts of **channel**-CPDL and **shared**-CPDL. These two versions are equivalent in expressive power (essentially since their program schemes are), and they lie in expressiveness between QDL with recursive procedures with parameters and QDL with array assignments. (This last comparison is restricted to infinite structures with a successor function.)

The third version, **count**-CQDL, is shown to be equivalent to CQDL, as the main message control is simulable on the first-order level.

In addition, we consider a restriction of the program system of **channel**-CQDL to programs with bounded concurrency. In the resulting system, named **channel-bc**-CQDL, the number of processes in a program is pre-defined, i.e., no new processes are allowed to be spawned. This version is shown to be equivalent in expressiveness to regular QDL.

The rest of the article is organized as follows. There are three main parts. The introductory Part A concludes with Section 2, in which we review CDL and define its semantics in a slightly different way than in [Pe1] (though the definitions are equivalent). Part B concerns the propositional level. Sections 3–5 define **channel**-CPDL, **shared**-CPDL, and **count**-CPDL, respectively. Section 6 contains some expressiveness results, and Section 7 concerns decidability of the validity problem. Part C deals with the first-order level. Section 8 presents **channel**-CQDL, **shared**-CQDL, and **count**-CQDL and Section 9 includes some related expressiveness results. Finally Section 10 contains a brief comparative discussion.

2. Preliminaries

2.1. *Concurrent* PDL

Let us start with an overview of the syntax and semantics of CPDL. The syntax of CPDL is just that of PDL, with the addition of a concurrency operator $\cap$ on programs. Thus, we have atomic formulas $\{P_i\}$, atomic programs $\{a_i\}$, and the construction rules:

(1) every P_i is a formula,

(2) if A and B are formulas, and α is a program, then $A \vee B$, $\neg A$, and $\langle \alpha \rangle A$ are formulas, and

(3) every a_i is a program,

(4) if α, β are programs and A is a formula, then $\alpha \cup \beta$, $\alpha \cap \beta$, $\alpha; \beta$, α^* and $A?$ are programs.

The semantics interprets formulas over models $\langle S, \pi, \rho \rangle$, where S is a set of states, π attaches a subset $\pi(P)$ of S to every atomic formula P, and ρ attaches a subset $\rho(a)$ of $S \times 2^S$ to every atomic program a. Intuitively, $(s, U) \in \rho(\alpha)$ for $s \in S$ and $U \subseteq S$ if α can be executed from s, to reach all states of U in parallel. Thus, the formula $\langle \alpha \rangle A$ holds in a state s iff there is a set $U \subseteq S$ s.t. $(s, U) \in \rho(\alpha)$, and each state in U satisfies A.

We extend π and ρ to all formulas and programs by the rules

$$\pi(A \vee B) = \pi(A) \cup \pi(B),$$
$$\pi(\neg A) = S - \pi(A),$$
$$\pi(\langle \alpha \rangle A) = \{s \mid \exists U((s, U) \in \rho(\alpha) \wedge U \subseteq \pi(A))\},$$

and

$$\rho(A?) = \{(s, \{s\}) \mid s \in \pi(A)\},$$
$$\rho(\alpha \cup \beta) = \rho(\alpha) \cup \rho(\beta),$$
$$\rho(\alpha \cap \beta) = \{(s, U) \mid \exists V, W((s, V) \in \rho(\alpha) \wedge (s, W) \in \rho(\beta) \wedge U = V \cup W)\},$$
$$\rho(\alpha; \beta) = \rho(\alpha) \cdot \rho(\beta),$$
$$\rho(\alpha^*) = \min\{R \mid R \subseteq S \times 2^S \wedge R = \rho(\text{true}?) \cup \rho(\alpha) \cdot R\},$$

where $R_1 \cdot R_2$ is defined (for any $R_1, R_2 \subseteq S \times 2^S$) as

$$\left\{(s, U) \mid \exists s_1, U_1, s_2, U_2, \ldots \left((s, \{s_1, s_2, \ldots\}) \in R_1 \right.\right.$$
$$\left.\left. \wedge \forall i (s_i, U_i) \in R_2 \wedge U = \bigcup_i U_i \right) \right\}.$$

The definition of $\langle \alpha \rangle A$ corresponds to the requirement that there exists a computation of α such that A holds in *every* one of its end-states. An alternative choice could be to make $\langle \alpha \rangle A$ hold whenever there exists a computation of α such that A holds in *some* of its end states. The approach we chose seems more appropriate for describing global correctness conditions required from (all the branches of) a program, and parallels the corresponding requirement on the branches of the computation of an alternating Turing machine [AKS]. Nevertheless, it should be clear that it is always possible to impose specific conditions on individual branches by including *tests* in appropriate points in the program, e.g., as in $\langle (\alpha; \varphi?) \cap \beta \rangle$ *true*.

In a subsequent paper [Pe3] we examine a more flexible approach to defining program schemes, based on the notion of tree-regular languages. One of the by-products of this approach is the capability of using reasonably general Boolean combinations of end-tests in formulas of the form $\langle \alpha \rangle A$.

See [Pe1] for a more detailed discussion of CPDL.

2.2. *Trecs*

We now give a slightly modified, yet equivalent construction for the interpretation ρ of programs, based on the notion of a *trec*, or a *tree-like computation*. This construction provides the technical basis for extending CDL for dealing with communication.

A *seq* (cf. [MeP]) is a (deterministic) PDL program containing no appearances of $\cup$ or $*$, i.e., a sequence of concatenated atomic programs and tests.

The role of trecs in concurrent programs is analogous to that of seq's in regular programs; both can be used to describe some specific, deterministic execution of a program. (Here, the term "deterministic" refers to the control structure of the program, and bears no implications on the interpretation of atomic programs.) A *trec*, or a *tree computation*, is a CPDL program containing no appearances of $\cup$ or $*$. Thus the set of trecs can be defined as the closure of atomic programs and tests under concatenation and $\cap$.

A trec α is said to be in *tree form* if it conforms to the following inductive definition.

(1) *true*? is a trec,

(2) if α, β are trecs, a is an atomic program and A is a formula, then $\alpha \cap \beta$, a; α and A?; α are trecs.

A trec in tree form can be represented as a tree whose arcs are labeled by the atomic programs and tests of the trec, and each $\cap$ corresponds to a split into two branches. The leaves of this tree correspond to the endpoints of the different parallel processes spawned by the trec. The transformation between trecs in tree form and the trees representing them is straightforward in both directions, and we will freely use both presentations.

Note 2.1. Every trec can be transformed into tree form. For instance, if α and β are in tree form, then $\alpha; \beta$ (not necessarily in tree form by itself) can be transformed into tree form by attaching the tree representing β to every leaf of the tree representing α. This corresponds to the repreated application of the transformation rule $(\theta \cap \gamma); \delta \Rightarrow \theta; \delta \cap \gamma; \delta$ to any sub-program of $\alpha; \beta$, whenever possible. Note also that this rule is valid for CPDL programs in general, and also with $\cup$ instead of $\cap$. However, if the δ is to the *left* of the brackets, then *both* transformations (either with $\cup$ or $\cap$) are invalid; hence a trec cannot be further decomposed into a set of completely separate parallel seq's.

It is well-known that the semantics of a (nondeterministic) sequential PDL program α can be defined on the basis of a set $\tau(\alpha)$ of (deterministic) seq's, describing the possible runs of α, so that $\rho(\alpha) = \bigcup_{\beta \in \tau(\alpha)} \rho(\beta)$. Analogously, the semantics of a (nondeterministic) concurrent program α can be based on a collection of (deterministic) trec's. Every program α is associated with a set of trecs $\tau(\alpha)$, representing all of its possible runs.

Note that the definition of τ serves merely as a transformation from one syntactic

object to another, and therefore gives meaning only to the compositional and structural connectives of the language (i.e., $\cap$, $\cup$, ; and $*$), and does not deal with the interpretation of any primitive actions in the model.

The sets $\tau(\alpha)$ are defined by

(1) $\tau(a_i) = \{a_i\}$, for atomic programs a_i,

(2) $\tau(A?) = \{A?\}$, for tests $A?$,

(3) $\tau(\alpha \cup \beta) = \tau(\alpha) \cup \tau(\beta)$,

(4) $\tau(\alpha \cap \beta) = \{\gamma \cap \delta \mid \gamma \in \tau(\alpha) \text{ and } \delta \in \tau(\beta)\}$,

(5) $\tau(\alpha; \beta) = \{\theta \mid \exists \gamma, \delta_1, ..., \delta_k (\gamma \in \tau(\alpha), \delta_1, ..., \delta_k \in \tau(\beta), \gamma \text{ has } k \text{ leaves, and } \theta \text{ is obtained by attaching each } \delta_i \text{ to one of the leaves of } \gamma)\}$,

(6) $\tau(\alpha^*)$ is the (minimal) set constructed by (precisely) the following rules:

(6.1) $true? \in \tau(\alpha^*)$.

(6.2) For every γ, δ and θ, if $\gamma \in \tau(\alpha^*)$, $\delta \in \tau(\alpha)$ and θ is obtained by attaching δ to one of the leaves of γ, then also $\theta \in \tau(\alpha^*)$.

Note that $\tau(\alpha)$ contains only trecs in tree form, and, in particular, for any trec α, $\tau(\alpha)$ contains (some) tree-form-equivalents of α.

Finally, we give the following characterization for $\rho(\alpha)$ in CPDL, equivalent to the original definition. First use that original definition to define $\rho'(\beta)$ for trecs β in tree form; then let $\rho(\alpha) = \bigcup_{\beta \in \tau(\alpha)} \rho'(\beta)$, for every program α.

Note 2.2. In all further references to the definition of $\rho(\alpha)$ in the semantics of CPDL, this characterization is implied.

2.3. **C-goto**-CPDL

We also need a different version of CPDL, defined and studied in [Pe3]. This version, named **c-goto**-CPDL, is based on representing programs in the form of a *concurrent goto scheme*, which is essentially a linear representation for a flowchart. A *propositional concurrent goto scheme* is a program composed of a sequence of labeled commands of the types

(ATOMIC) l: a_i,

(TEST) l: $P?$,

(N-GOTO) l: *goto* l' *or* l'',

(PAR) l: *goto* l' *and* l'',

where P is a formula in **c-goto**-CPDL, and a_i is an atomic command.

The semantics of **c-goto**-CPDL is based on models similar to those of CPDL. Informally, the semantics of the (N-GOTO) command is that of a nondeterministic branch, and the semantics of a (TEST) is that of a guard, i.e. it can be passed only if the formula P evaluates to true in the present state, otherwise the run aborts.

When a process reaches a command of type (PAR), it splits into two parallel processes, identical by their state. (Giving this "real life" interpretation, we may say that the memory locations held by the original process are duplicated, and each of the two new processes receives an identical private copy.) Now one of the processes proceeds to execute the command labeled l', and the other proceeds to l'', and from now on the two processes remain separate and independent, with no connection whatsoever.

A detailed formal definition of the semantics is given [Pe3], and will not be repeated here. We note for future use

PROPOSITION 2.1 [Pe3]. *The validity problem for* **c-goto**-CPDL *can be decided in nondeterministic double exponential time.*

2.4. *Concurrent* QDL

Let us now review also the quantified (first-order) language CQDL, which relates to QDL just as CPDL relates to PDL.

The language of CQDL is defined analogously to CPDL, on the basis of first-order logic (with equality) (cf. [H1]). Thus, the atomic formulas are predicates $P(\bar{\sigma})$ where $\bar{\sigma}$ is a tuple of terms. The programs are defined just like their propositional counterparts, except that the atomic operations are chosen to be simple assignments $x_i \leftarrow \sigma$. Programs and formulas use a tuple of variables $\bar{x} = \{x_1, ..., x_n\}$, and the assignments and tests refer to some fixed signature; σ is a term over the signature involving variables from $\bar{x}$.

The semantics is based on a first-order structure $\mathscr{A} = \langle \mathscr{D}, P_1, ..., f_1, ... \rangle$ with a domain $\mathscr{D}$ and a collection of predicates P_i and functions f_i, corresponding to the signature of the language. The set of states S associated with such structure is the set of possible interpretations for the variable set $\bar{x}$ used in the scheme; every possible assignment of values from $\mathscr{D}$ to variables in $\bar{x}$ corresponds to a state. The structure induces an interpretation for terms σ in every state, and also an interpretation of the atomic formulas; the truth-value of $P(\bar{\sigma})$ in a state s is determined by the value of $P(\bar{\sigma}_s)$ in the structure, where $\bar{\sigma}_s$ is the evaluation of $\bar{\sigma}$ in the state s.

A subset $\pi(\varphi)$ of S is attached to every formula φ, and a subset $\rho(\alpha)$ of $S \times 2^S$ to every program α. The values of $\pi(\varphi)$ are determined just as in ordinary QDL, with the obvious modification

$$\pi(\langle \alpha \rangle \varphi) = \{s \mid \exists V((s, v) \in \rho(\alpha) \wedge V \subseteq \pi(\varphi))\},$$

and ρ is defined just as in CPDL, with an additional rule for assignments

$$\rho(x_i \leftarrow \sigma) = \{(s, s[\sigma_s/x_i]) \mid s \in S\},$$

where $s[\sigma_s/x_i]$ represents a state s' similar to s except that x_i is interpreted in s' as σ_s, the evaluation of the term σ in s.

PART B: THE PROPOSITIONAL LEVEL

3. **Channel**-CPDL

A run of a CPDL program (or trec) can be described by a tree in which there are no connections between different branches. The run starts at a certain common state s, but once the program splits into two or more parallel processes, these processes advance separately, with no synchronization or connection whatsoever.

The semantic world of concurrent programs with communication can no longer have this simple form; there must be some connection between the different branches. Therefore it is advantageous to consider *super-states*, representing *cuts* in the computation tree, and the semantic definitions have to refer simultaneously to all states in the cut and all programs operating in the different branches. For example, Fig. 3.1 shows a sequence of super-states representing a run on some computation tree.

We want to allow a full nondeterminism of concurrency, in the sense that any "timing shuffle" of the separate branches is allowed (unless it violates the synchronization restrictions imposed by the communication). Communication between two parallel processes is enabled in **channel**-CPDL by means of channels, as is done in several concurrent languages. The notation we use resembles that of CCS [M1].

We proceed to give a precise definition of the syntax and semantics of **channel**-CPDL.

Syntax of **channel**-CPDL

The syntax is based on CPDL, with the following additions: The language has a collection of channels $\{C_i\}$. In addition to atomic programs and tests, the following atomic *communication operations* are allowed: $C!0$ (resp. $C!1$) reading "transmit 0 (resp. 1) over channel C", and $C?0$ (resp. $C?1$) reading "receive 0 (1) over channel C".

Semantics of **channel**-CPDL

Informally, communication is carried out in the following manner. When a process reaches the command $C!0$, it has to wait for some parallel process to execute $C?0$. These two operations happen simultaneously, and then each process continues separately. (A similar procedure occurs for $C!1$ and $C?1$.) Both processes

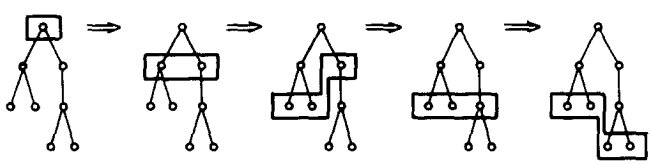

FIGURE 3.1

remain, after executing a communication command, in the state they were in before communication took place. Thus, actually, communicating can be viewed as the synchronization of tests, or "guards," in the two processes.

Nevertheless, it is clear that these commands can be used by a process in order to transfer, for instance, the value of a predicate P in its current state (by executing $P?; C!1 \cup (\neg P)?; C!0$) and another process can make a decision on the basis of such a message (by executing $C?0; \alpha \cup C?1; \beta$).

A formal definition of the semantics of **channel**-CPDL is rather complicated, compared to that of CPDL, as a result of having to deal simultaneously with super-states instead of handling each branch separately. For instance, $\rho(\alpha \cap \beta)$ can be defined in CPDL by defining first $\rho(\alpha)$ and $\rho(\beta)$ separately, but in **channel**-CPDL this definition cannot be split without introducing additional semantical objects. In fact, if there are common channels between α and β, it may be that α or β are not executable at all, once separated. Thus the semantic rules must also consider several parallel processes, or a *super-process.*

The model provides an interpretation $\rho(a) \subseteq S \times S$ for atomic programs a, thus we limit ourselves to sequential models (cf. [Pe1]). This definition now has to be extended to super-processes.

We use the following notation. A super-state is denoted by $\bar{s} = [s_1 \mid \cdots \mid s_n]$ (when $n = 1$ we identify s_1 with $[s_1]$, for simplicity). When $\bar{s}$ is known we may refer to a portion of it, $[s_i \mid \cdots \mid s_{i+k}]$, as $\bar{s}(i, i+k)$. Two super-states may be combined to form a larger super-state. This is denoted by $[s_1 \mid \cdots \mid s_n] \mid [q_1 \mid \cdots \mid q_m] = [s_1 \mid \cdots \mid s_n \mid q_1 \mid \cdots \mid q_m]$. The width of $\bar{s}$, denoted $|\bar{s}|$, is the number n of parallel states in $\bar{s}$. Similarly, a super-process is denoted by $\bar{\alpha} = [\alpha_1 \mid \cdots \mid \alpha_n]$ and we may describe portions of it by $\bar{\alpha}(i, i+k)$. Again we identify α_1 with $[\alpha_1]$. For a super-process $\bar{\alpha} = [\alpha_1 \mid \cdots \mid \alpha_n]$, $\rho(\bar{\alpha}) \subseteq S^n \times (\bigcup_{i \geqslant n} S^i)$, i.e., $\rho(\bar{\alpha})$ consists of tuples $(\bar{s}, \bar{s}')$ with $|\bar{s}| = n$ and $|\bar{s}'| \geqslant n$.

The definition of $\rho(\bar{\alpha})$ for every super-process $\bar{\alpha}$ follows the lines of the corresponding definition for CPDL in Section 2.2. In general, we retain the distinction between the compositional/structural connectives of the language and the primitive operations. As far as the first ones go, we keep the tree-like nature of our basic deterministic programs—the trecs, and the decomposition of general programs into trecs. Hence, we still have the transformations described in Note 2.1, including

(1) $(\theta \cap \gamma); \delta \Rightarrow \theta; \delta \cap \gamma; \delta$ and

(2) $(\theta \cup \gamma); \delta \Rightarrow \theta; \delta \cup \gamma; \delta$.

We first define the set of trecs $\tau(\alpha)$ associated with each program α. This definition goes exactly as in Section 2.2, modified of course to account also for atomic communication operations.

Next we define $\rho(\bar{a})$ for a super-process $\bar{\alpha}$ consisting only of trecs in tree form. At this stage, the semantic interpretation of the channel communication operations imposes some synchronization connections between separate branches of the trecs, so these branches are no more independent and the trecs lose their tree-like nature.

(1) $\rho(true?) = \{(s, s) \mid s \in S\}$.

(2) If $\alpha_i = a; \beta$, $(s_i, s') \in \rho(a)$
and $(\bar{s}(1, i-1) \mid s' \mid \bar{s}(i+1, n), \bar{r}) \in \rho(\bar{\alpha}(1, i-1) \mid \beta \mid \bar{\alpha}(i+1, n))$
then $(\bar{s}, \bar{r}) \in \rho(\bar{\alpha})$.

(3) If $\alpha_i = C!1; \beta_1$, $\alpha_j = C?1; \beta_2, j > i$
and $(\bar{s}, \bar{r}) \in \rho(\bar{\alpha}(1, i-1) \mid \beta_1 \mid \bar{\alpha}(i+1, j-1) \mid \beta_2 \mid \bar{\alpha}(j+1, n))$
then $(\bar{s}, \bar{r}) \in \rho(\bar{\alpha})$.
Similarly when $j < i$, and when the message is 0 on both sides.

(4) If $\alpha_i = A?; \beta$, $s_i \in \pi(A)$
and $(\bar{s}, \bar{r}) \in \rho(\bar{\alpha}(1, i-1) \mid \beta \mid \bar{\alpha}(i+1, n))$
then $(\bar{s}, \bar{r}) \in \rho(\bar{\alpha})$.

(5) If $\alpha_i = \beta \cap \gamma$
and $(\bar{s}(1, i-1) \mid s_i \mid s_i \mid \bar{s}(i+1, n), \bar{r}) \in \rho(\bar{\alpha}(1, i-1) \mid \beta \mid \gamma \mid \bar{\alpha}(i+1, n))$
then $(\bar{s}, \bar{r}) \in \rho(\bar{\alpha})$.

For every $\bar{\alpha}$, $\rho(\bar{\alpha})$ contains precisely those pairs $(\bar{s}, \bar{r})$ introduced by the above rules.

The definition of ρ is extended to any super-process $\bar{\alpha}$ by letting

$$\rho([\alpha_1 \mid \cdots \mid \alpha_n]) = \bigcup_{\substack{\beta_i \in \tau(\alpha_i) \\ 1 \leqslant i \leqslant n}} \rho([\beta_1 \mid \cdots \mid \beta_n]).$$

Finally, the interpretation of formulas has to be modified accordingly, by letting

$$\pi(\langle \alpha \rangle A) = \{s' \mid \exists \bar{s}(\bar{s} = [s_1 \mid \cdots \mid s_n], (s', \bar{s}) \in \rho(\alpha) \text{ and } \forall i, 1 \leqslant i \leqslant n(s_i \in \pi(A)))\}.$$

Discussion. The definition of program semantics here differs from that given to concurrent programs without communication. Actually, here we interpret a program computation as leading from a single state to a *multiset* of states. This may lead to a serious difference in the interpretation of certain programs. For example, let $tt = true? \cap true?$, $\alpha = tt; a$ and $\rho(a) = \{(s_0, s_1), (s_0, s_2)\}$. Then by our previous rules $\rho(tt) = \rho(true?)$, and $\rho(\alpha) = \{(s_0, \{s_1\}), (s_0, \{s_2\})\}$, while according to the multiset interpretation $\rho_{\text{mult}}(\alpha) = \{(s_0, [s_1 \mid s_1]), (s_0, [s_1 \mid s_2]), (s_0, [s_2 \mid s_1]), (s_0, [s_2 \mid s_2])\}$, meaning that it is possible for α to split and reach two different states.

We chose the "set" interpretation for concurrent programs without communication for its simplicity and compatibility with the definitions of game logic. When communication is involved, this does not seem to work anymore. However, we should observe that this choice of semantics for schemes does not affect the interpretation of *formulas* in the logics. The reason is that, as can be easily shown,

(1) $\rho(\alpha) \subseteq \rho_{\text{mult}}(\alpha)$ for every program α (iditifying a multiset U with the set $set(U)$ consisting precisely of its elements), and

(2) $(s, U) \in \rho_{\text{mult}}(\alpha) \Rightarrow \exists U'(U' \subseteq \text{set}(U), (s, U') \in \rho(\alpha))$ for every program α.

These two observations serve to prove that for any formula A, $\pi(A) = \pi_{\text{mult}}(A)$. This holds on both the propositional and first-order levels.

As for the interpretation of formulas, one should note that by this semantics, two programs α and β in separate contexts (say, in $\langle \alpha \rangle A$ and $\langle \beta \rangle B$) are still totally disconnected and no communication is possible between them. Furthermore, even the attached formula A has no way of referring to the channels of α, and the same applies even to tests in α. This means, for instance, that valid CPDL axioms such as $\langle \alpha \cap \beta \rangle A \equiv \langle \alpha \rangle A \wedge \langle \beta \rangle A$ and $\langle \alpha; \beta \rangle A \equiv \langle \alpha \rangle \langle \beta \rangle A$ no longer hold in **channel**-CPDL. For example, $\langle C!0 \cap C?0 \rangle$ *true* is always true, while $\langle C!0 \rangle$ *true* $\wedge \langle C?0 \rangle$ *true* is always false.

The following examples demonstrate the expressive power gained by the introduction of communication.

EXAMPLE 3.1. *Leaf counting.* Considering models in the form of finite full binary *a/b* trees, the formula $even_1$ addresses exactly those models in which there is an even number of leaves satisfying P (see Fig. 3.2).

$$even_1: \quad \langle ((a \cap b)^*; leaf?; ((\neg P)? \cup (P?; C!1))) \cap (C?1; C?1)^* \rangle \; true,$$

where

$$leaf: \quad \neg \langle a \cup b \rangle \; true.$$

The main program of $even_1$ splits into several parallel processes, one for each branch of the tree. Each of these processes, upon reaching a leaf, tests P, and sends a message if P is true. A separate process receives these messages and keeps track of the parity of their number.

In order to account also for partial binary trees (i.e., in which a node may have one child), replace the sub-program $(a \cap b)^*$ with

$$\begin{aligned} &((\langle a \rangle \; true \wedge \langle b \rangle \; true)?; (a \cap b) \\ &\quad \cup (\langle a \rangle \; true \wedge \neg \langle b \rangle \; true)?; a \\ &\quad \cup (\neg \langle a \rangle \; true \wedge \langle b \rangle \; true)?; b)^*. \end{aligned}$$

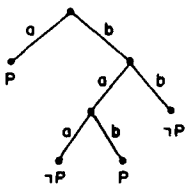

FIG. 3.2. A finite full binary *a/b* tree satisfying $even_1$.

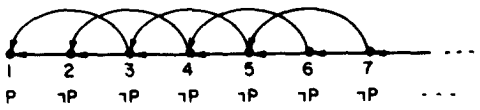

FIGURE 3.3

Applied to a finite binary *a/b dag*, the program counts the number of different paths leading to leaves satisfying P. Thus, consider a model as in Fig. 3.3, where in each state exactly one of the arcs leaving it is denoted by a, and one by b. In such a model, $\pi(even_1) = \{3k \mid k \geqslant 1\}$, since the number of different paths from a state n to the end is exactly the nth Fibonacci number, which is even for $n = 3k$, $k \geqslant 1$.

These examples demonstrate one function of communication, namely, sending messages and "final reports" from independent subprocesses to a main "counting" process. However, **channel**-CPDL can provide for the other functions as well. The following example demonstrates the use of synchronization.

EXAMPLE 3.2. *Post correspondence problem.* This example is based on the ideas used in [HPS] to show the undecidability of PDL + $\{\alpha^i\beta\gamma^i \mid i \geqslant 0\}$, by reducing the Post correspondence problem (PCP) to its satisfiability problem.

The models concerned here are each in the form of a finite path $p_{\mathcal{M}} = (s_0, s_1, \ldots, s_m)$ connected by an atomic program a, with an atomic predicate P interpreted over the states of the model. Such a model $\mathcal{M}$ represents a word $z_{\mathcal{M}} = \sigma_1 \cdots \sigma_m$; for each $1 \leqslant i \leqslant m$, $\sigma_i = 1$ if $s_i \models P$, otherwise $\sigma_i = 0$.

Let $\bar{x} = (x_1, \ldots, x_n)$, $\bar{y} = (y_1, \ldots, y_n)$ be a given instance of PCP. We construct a **channel**-CPDL formula $correspond_{\bar{x},\bar{y}}$ which is satisfiable in the model $\mathcal{M}$ iff the represented word $z_{\mathcal{M}}$ is a solution to the given PCP instance $(\bar{x}, \bar{y})$.

The formula $correspond_{\bar{x},\bar{y}}$ employs two processes, one to "guess" and verify an x-partition of the word, and the other for the y-partition. The processes make use of channels C_i $(1 \leqslant i \leqslant n)$ in order to force the simultaneous application of words x_i and y_i.

The formula $correspond_{\bar{x},\bar{y}}$ is constructed as follows. For any word $x_i = w_{i,1} \cdots w_{i,n_i}$ construct a program $\alpha_i = a_{i,1}; \cdots; a_{i,n_i}; C_i!1$, where

$$a_{i,j} = \begin{cases} a; P?, & w_{i,j} = 1 \\ a; (\neg P)?, & w_{i,j} = 0. \end{cases}$$

For any word $y_i = v_{i,1} \cdots v_{i,n_i}$ construct a program $\beta_i = b_{i,1}; \cdots; b_{i,n_i}; C_i?1$, with $b_{i,j}$'s defined analogously by the $v_{i,j}$'s.

Let

$$\alpha = \Big(\bigcup_{1 \leqslant i \leqslant n} \alpha_i \Big)^* \quad \text{and} \quad \beta = \Big(\bigcup_{1 \leqslant i \leqslant n} \beta_i \Big)^*.$$

Finally define

$$correspond_{\bar{x},\bar{y}}: \quad \langle \alpha \cap \beta \rangle \, leaf.$$

correspond$_{\bar{x},\bar{y}}$ states that α and β can be executed in parallel on the path $p_{\mathcal{M}}$, which implies that $p_{\mathcal{M}}$ can be decomposed into k parts corresponding to a sequence of k words x_i, and, alternatively, into k parts corresponding to k words y_i, with matching indices (as verified through the appropriate channels). This means that the model indeed represents a solution to the given PCP instance.

4. **Shared**-CPDL

The second system to be discussed is **shared**-CPDL, in which we allow the use of shared Boolean variables.

Syntax of **shared**-CPDL

In addition to the basic syntax of CPDL, the language also has a set of Boolean variables $\{X_i\}$. For any X_i, the language allows $X_i = 0$ and $X_i = 1$ as atomic formulas, and $X_i \leftarrow 0$ or $X_i \leftarrow 1$ are atomic programs.

These variables thus resemble the Boolean variables proposed in [A], forming B-PDL as an extension of PDL. However, while B-PDL is shown to be essentially no more powerful than PDL, the shared variables of **shared**-CPDL increase its power considerably. Actually, it is possible to add Boolean variables to concurrent schemes as *local* variables, with a private set of variables for every process [Pe3]. This yields a Boolean version of CPDL which is essentially equivalent to CPDL, in the sense of [A].

Semantics of **shared**-CPDL

The evaluation of the shared variables is not a part of any state s_i, but an independent part of the super-state. Thus a super-state $\hat{s}$ consists of two parts: a vector $\bar{s} = [s_1 | \cdots | s_n]$ of states (where each s_i contains a separate valuation for the ordinary predicates), and a valuation I_s: $\{X_i\} \to \{0, 1\}$ for the shared variables. This means that the value of X is identical in all of the sub-states s_i. In particular, an assignment into a shared variable in one of the processes has to change I_s.

This definition alone, coupled with the nondeterministic semantics of concurrency in CPDL, does not allow an intelligible use of communication. We must allow also some mechanism providing the feature of mutual exclusion. This may be achieved, for instance, by means of a *test-and-set* command, $TS(X)$, meaning the indivisible execution of $X = 0?; X \leftarrow 1$ (i.e., with no interference of other processes allowed between the two steps).

The appropriate semantic rules are:

Semantic Rule for $X \leftarrow 0$

if $\alpha_i = X \leftarrow 0; \beta, \hat{s} = (\bar{s}, I_s)$
and $((\bar{s}, I_s[0/X]), \hat{r}) \in \rho(\bar{\alpha}(1, i-1) \,|\beta|\, \bar{\alpha}(i+1, n))$
then $(\hat{s}, \hat{r}) \in \rho(\bar{\alpha})$.

(Here, by $I_s[0/X]$ we mean an interpretation I' that is equivalent to I_s, except that X is interpreted as 0. Similar notation will be used in several different contexts in the sequel.)

Semantic Rule for $TS(X)$

$$\begin{aligned}&\text{if } \alpha_i = TS(X);\ \beta,\ \hat{s} = (\bar{s}, I_s),\ I_s(X) = 0\\ &\text{and } ((\bar{s}, I_s[1/X]),\ \hat{r}) \in \rho(\bar{\alpha}(1, i-1)\ |\beta|\ \bar{\alpha}(i+1, n))\\ &\text{then } (\hat{s}, \hat{r}) \in \rho(\bar{\alpha}).\end{aligned}$$

These rules replace part (3) in the definition of the semantics of **channel**-CPDL. The rest of the definition remains the same.

5. **count**-CPDL

As will be shown later, the powerful capabilities of **channel**-CPDL render its validity problem undecidable. This follows, for instance, from the capability of synchronization, as demonstrated in Example 3.2. Such behaviour (on the propositional level) raises the interesting question of finding a "less powerful" version of communication, with "more desirable" properties. This motivates the consideration of a limited version of **channel**-CPDL, in which communication is non-blocking, i.e., the messages sent through channels are not synchronized, or "time-stamped," in any way. When a process wishes to send a message on some channel C, it simply does so and proceeds with its activities, without worrying about the existence of a receiving process. Similarly, whenever a process wishes to receive a mesage, it "accepts" it and proceeds, even if no process has sent it yet. The only requirement is that in the end of the whole computation, there will be an appropriate matching for "sent" and "received" messages (including from a process to itself). This situation can be thought of also as being handled by a control mechanism based on an (unordered) "infinite buffer," which keeps and matches different "send" and "receive" requests of messages.

We present this system in a slightly cleaner form, through the notion of *counts*. We define two logics named **count**-CPDL and **u-count**-CPDL. These logics do not provide all functions of communication discussed earlier. In particular, separate processes are independent and ignorant of each other; they are not synchronizable, and cannot receive messages.

The only sense in which there is communication in **count**-CPDL and **u-count**-CPDL is that processes can send reports, or messages, to an imaginary main control process. This control process has, in principle, no means of identifying the source of messages, or the order in which they were sent; it can only count the number of messages it has received of each type, according to its specifications.

We distinguish between several types of counting. A particularly powerful type is that of *unbounded counting*. This type appears in **u-count**-CDPL but not in **count**-

CPDL. Later we see that this powerful capability renders **u-count**-CPDL undecidable. (In fact, the properties of unbounded counting and synchronization are independent of each other, and each alone causes undecidability.)

Syntax of **count**-CPDL

The syntax of **count**-CPDL is similar to that of CPDL, with the following changes. The languages has a set of *messages* $\{M_i\}$. Any message M_i is allowed as an atomic program (in addition to the usual atomic programs and tests). Given a program α containing appearances of messages $M_1,\ldots, M_k$, a specification *spec* as described below and a formula A, $\langle\alpha\rangle_{spec} A$ is also a formula.

The subscript *spec* stands for a collection of *counting specifications* of the following three types:

(S1) modular counting: $\#M_i = l \pmod t$,

(S2) full counting: $\#M_i = l$,

(S3) modular comparison: $\#M_i = \#M_j \pmod t$.

A fourth type of counting specifications is

(S4) full comparison: $\#M_i = \#M_j$.

The language resulting from allowing also full comparisons (S4) is **u-count**-CPDL.

Semantics of **count**-CPDL

The intended meaning of the formula $\langle\alpha\rangle_{spec} A$ is that the number of messages sent by α has to match the specification of *spec*. For example, if $spec = \{\#M_1 = \#M_2 \pmod 5\} \cup \{\#M_3 = 7\}$, then the formula claims that some trec β of α can be executed so that A holds at the end of each branch, the number of messages M_1 and M_2 sent in β is equal, modulo 5, and the number of M_3 messages sent in β is 7. (Here, again, the definition of trecs has to be extended to include also messages as atomic steps.)

A formal definition of the semantics now follows. The interpretation $\rho'(\alpha)$ of a trec α in tree form is just as in CPDL. In addition let $\rho'(M_i) = \rho'(true?)$, for any message M_i.

For any trec α, let $\#(M_i, \alpha)$ denote the number of occurrences of M_i in α.

Let *spec* be a given set of specifications. For every trec α, let $spec_\alpha$ denote the set of statements obtained from *spec* by replacing any appearance of $\#M_i$ with $\#(M_i, \alpha)$. Let $\tau(spec)$ denote the collection of trecs conforming to the specifications of *spec*,

$$\tau(spec) = \{\alpha \mid \alpha \text{ is a trec, all statements in } spec_\alpha \text{ are true}\}.$$

Given any program α, restrict $\tau(\alpha)$ to trecs satisfying the requirements of *spec* by letting

$$\tau(\alpha, spec) = \tau(\alpha) \cap \tau(spec).$$

For any program α let $\rho(\alpha, spec) = \bigcup_{\beta \in \tau(\alpha, spec)} \rho'(\beta)$. Finally define $\pi(\langle\alpha\rangle_{spec} A)$ as in CPDL, only using $\rho(\alpha, spec)$ instead of $\rho(\alpha)$.

The same definition applies to the semantics of **u-count**-CPDL too, with *spec* understood to include also specifications of type (S4).

Despite its obvius limitations, **count**-CPDL can express several properties which we suspect to be otherwise inexpressible in CPDL.

EXAMPLE 5.1. *Leaf counting.* The **channel**-CPDL formula $even_1$, from Example 3.1, can be written also in **count**-CPDL, as

$$even_2: \qquad \langle (a \cap b)^*; leaf?; ((\neg P)? \cup P?; M)\rangle_{\{\# M = 0 \pmod 2\}}\ true.$$

This formula works just as does $even_1$, except that the parity counting is done implicitly, and not by an explicitly defined process.

A slightly different formula of interest is

$$one\text{-}mod\text{-}three: \qquad \langle (a \cap b)^*; leaf?; M\rangle_{\{\# M = 1 \pmod 3\}}\ true.$$

Referring to some state in a finite binary a/b tree (or dag), this second formula asserts that the number of different paths leading from the state to leaves in the tree (dag) is 1, modulo 3. In particular, in the case of a full and balanced binary tree this means that the state is at an even height, considering leaves to be at height 0 (this specific case can of course be handled in PDL too, by a simpler formula).

The next example is a **u-count**-CPDL formula demonstrating the power of unbounded counting.

EXAMPLE 5.2. *A grid.* This example concerns infinite full binary a/b dags. Given any **u-count**-CPDL formula ψ, the formula $grid(\psi)$ asserts that the model behaves as an a/b grid (see Fig. 5.1) with respect to ψ. The formula $grid(\psi)$ holds in a state s iff for every $m, n \geqslant 0$ and for every two states s_1, s_2 reachable from s by seq's

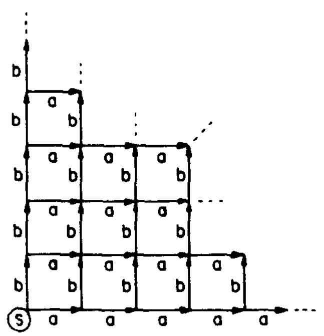

FIG. 5.1. An a/b grid.

containing m a's and n b's, $s_1 \in \pi(\psi) \Leftrightarrow s_2 \in \pi(\psi)$. This formula is expressable in **count**-CPDL as

$$\neg\langle((a; M_1 \cup b; M_2)^*; \psi?) \cap ((a; M_3 \cup b; M_4)^*; (\neg\psi)?)\rangle_{spec}\, true,$$

where

$$spec = \{\#M_1 = \#M_3,\ \#M_2 = \#M_4\}.$$

For future purposes, let *grid*-PDL be the language

$$\text{PDL} \cup \{grid(\psi) \mid \psi \in prop.\ calculus\}.$$

Finally we show some properties of **count**-CPDL.

DEFINITION. A **u-count**-CPDL formula is *singly specified* iff in any subformula $\langle\alpha\rangle_{spec} A$, any message M_i participates in at most one specification statement of *spec*.

LEMMA 5.1. *For any* **u-count**-CPDL *formula A there is an equivalent singly specified formula A'.*

Proof. Let $\langle\alpha\rangle_{spec} B$ be a subformula of A, and assume that the message M_i appears in k different statements of *spec*. Replace every occurrence of M_i in α with $M_i^1; \ldots; M_i^k$, where M_i^j, $1 \leqslant j \leqslant k$ are new message symbols, and modify the k specification statements involving M_i so that each will concern exactly one distinct M_i^j. ∎

Consequently, we will freely assume formulas to be singly specified, whenever convenient.

We now claim that the specification type (S3) (modular comparisons) is redundant in the presence of type (S1) (modular counting).

LEMMA 5.2. *For every* **u-count**-CPDL *formula involving modular comparison statements there is an equivalent formula with no such statements. (The same is true for* **count**-CPDL, *too.)*

Proof. Construct the equivalent formula inductively, based on the following main step. Assume that the subformula $A = \langle\alpha\rangle_{spec} B$ contains a statement *stat*: $\#M_i = \#M_j \pmod t$. Let $spec' = spec - \{stat\}$, and $spec_l = \{\#M_i = l \pmod t,\ \#M_j = l \pmod t\}$, for $0 \leqslant l \leqslant t-1$, and replace A by $\bigvee_{0 \leqslant l \leqslant t-1}(\langle\alpha\rangle_{spec' \cup spec_l} B)$. ∎

6. *Expressiveness Results*

Comparisons of expressive power are based on the following definitions. Two formulas φ, ψ are equvalent ($\varphi \equiv \psi$) if $\pi(\varphi) = \pi(\psi)$ in every model $\mathcal{M}$. For two logics

L_1, L_2 interpretable over compatible models, we say that $L_1 \leqslant L_2$ iff for every formula $\varphi \in L_1$ there is an equivalent formula $\psi \in L_2$. $L_1 = L_2$ iff $L_1 \leqslant L_2$ and $L_2 \leqslant L_1$. $L_1 < L_2$ iff $L_1 \leqslant L_2$ and not $L_2 \leqslant L_1$.

When we try to compare the logics based on channels and shared variables with each other, we run into the problem of their incomparable models. Strictly speaking, **channel**-CPDL formulas cannot refer to the variables of **shared**-CPDL. However, comparisons can be made in a way similar to the comparisons between the Boolean and propositional versions of PDL and CPDL [A, Pe3], i.e., by ignoring the "shared" part of the semantics, and considering the "state" part alone. Such comparisons are dependent on the initial values of the shared variables. Thus, every **shared**-CPDL formula can be simulated by a set of **channel**-CPDL formulas, one formula for each possible assignment I_s of initial values to the variables. The interpretation of any of the **channel**-CPDL formulas is identical to the "state part" of the original formula, assuming that initial interpretation.

Let L_1 be a logic with shared variables and L_2 be a propositional logic. We define the following notions. For a formula $A \in L_1$ involving shared variables $\bar{X} = (X_1, ..., X_n)$, a formula $B \in L_2$ and a Boolean tuple $\bar{b} = (b_1, ..., b_n)$, we say that $A \equiv_{\bar{b}} B$ ($\bar{b}$ *equivalence*) iff for every state s and every interpretation I, in every shared model, $s \in \pi(B) \Leftrightarrow (s, I[\bar{b}/\bar{X}]) \in \pi(A)$. $L_1 \leqslant_I L_2$ iff for every formula $A \in L_1$ (involving $\bar{X}$) and for every Boolean tuple $\bar{b}$, there is a formula $A_{\bar{b}} \in L_2$ s.t. $A \equiv_{\bar{b}} A_{\bar{b}}$. $L_1 =_I L_2$ (*I-equivalence*) iff $L_1 \leqslant_I L_2$ and $L_2 \leqslant L_1$.

It is easy to see that **shared**-CPDL can simulate the channel communication of **channel**-CPDL, in the usual way. A channel C can be represented by a tuple of three shared variables, $\langle X_c, M_c, G_c \rangle$, where X_c stores the value of the transferred message, M_c stores the *mode* of the channel (with 0 meaning the channel is free, and 1 meaning a mesage was broadcasted, and the channel awaits a receiving process), and G_c implements the guard ensuring mutual exclusion.

Thus a command $C!0$ can be simulated by

$$TS(G_c); M_c = 0?; X_c \leftarrow 0; M_c \leftarrow 1; G_c \leftarrow 0; M_c = 0?$$

and similarly, for $C?0$,

$$TS(G_c); M_c = 1?; X_c = 1?; M_c \leftarrow 0; G_c \leftarrow 0.$$

Note that we must delay the transmitting process until its message is received by someone, otherwise it might continue its run, and eventually read its own message.

A formula $\langle \alpha \rangle A$ in **channel**-CPDL can now be translated into $\langle init; \alpha' \rangle (proper\text{-}end \wedge A)$, where *init* sets initial zero vaues to all G_c and M_c variables, α' is the same as α except that communication commands are replaced as above, and *proper-end* checks all that M_c variables end with zero values.

Hence we have shown

LEMMA 6.1. **channel**-CPDL $\leqslant$ **shared**-CPDL.

The above simulation involves a subtle point which demonstrates a certain weakness of CPDL, namely, its inapropriateness for handling deadlocks. During the simulation of a channel communication operation, a "bad timing" of some **shared**-CPDL processes might cause a deadlock. However, such a possible run does not affect the interpretation ρ of the program, as it does not result in any legal "start–end" pair of states; hence, the semantics ignores such a run, much in the same way that it ignores runs which fail due to a false test (e.g., which try to test for $P?$ in a state where P is false). See also the discussion in Section 10.

In the other direction we can show

PROPOSITION 6.2. **shared**-CPDL $\leqslant_l$ **channel**-CPDL.

Proof. Let A be a **shared**-CPDL formula containing occurrences of shared variables $\bar{X} = (X_1, ..., X_n)$. The formulas $A_{\bar{b}}$ for any Boolean tuple $\bar{b}$ are constructed by structural induction on A.

$\bar{X}$-free A: Let $A_{\bar{b}} = A$.

$A = B \vee D$, $\neg B$: Trivial.

$A = X_i$: Let $A_{\bar{b}} = b_i$ (or, actually, the corresponding *true* or *false*).

$A = \langle \alpha \rangle B$: First transform A into $\langle \beta \rangle$ *true* where $\beta = \alpha; B?$. By the inductive hypothesis, for every test $E?$ in β and for every interpretation $\bar{b}$ there is a **channel**-CPDL formula $E_{\bar{b}}$ as required.

In the simulating program we have a separate, dedicated process $simul_x^b$ for every shared variable X, simulating X under the assumption that X is initialized by b. This process uses two channels, C_x and G_x, to conduct the assignments/tests and the TS commands, respectively. The program β is transformed into β' by the following changes:

(1) Occurrences of $X \leftarrow 1$ ($X \leftarrow 0$) are replaced by $C_x!1$ ($C_x!0$), respectively.

(2) Occurrences of $TS(X)$ are replaced by $G_x!1$.

(3) Any test $E?$ is replaced by $\bigcup_{\bar{b}} (C_{x_1}?b_1; ...; C_{x_n}?b_n; E_{\bar{b}}?)$.

For each shared variable X construct the following programs.

$$
\begin{aligned}
represent_x^1: &\quad (C_x!1)^* \\
represent_x^0: &\quad (C_x!0)^*; (true? \cup G_x?1; (C_x!1)^*) \\
sim_x: &\quad (C_x?1; represent_x^1 \cup C_x?0; represent_x^0)^* \\
simul_x^1: &\quad represent_x^1; sim_x \\
simul_x^0: &\quad represent_x^0; sim_x
\end{aligned}
$$

Finally, for every Boolean tuple $\bar{b}$, let

$$
A_{\bar{b}} = \left\langle \left(\bigcap_{1 \leqslant i \leqslant n} simul_{x_i}^{b_i} \right) \cap \beta' \right\rangle true.
$$

Note that we do not care to prevent other processes from changing the values of the shared variables while we check the pre-conditions of a test, in a program segment $(C_{x_1}?b_1;\ldots;C_{x_n}?b_n;E_\beta?)$. (This could be done by using semaphores to make this whole segment a critical section.) The reason is that this segment has no communication other than with the $simul_x$ processes, and therefore it has no synchronization constraints on it; if the test can be executed successfully provided some other process changes the value of X_i *during* the execution of the program segment, then it can be executed also with the other process changing X_i *before* the test segment started. ▮

Consequently we have

COROLLARY 6.3. **shared**-CPDL $=_I$ **channel**-CPDL.

This yields two possible alternatives for full comparison of the two systems, each of which requiring a change in one of the logics, to form a common basis.

The first alternative is to consider essentially **shared**-CPDL models. This requires us to extend the semantics of **channel**-CPDL so as to allow it to recognize shared variables within atomic predicates $X=0/1$. Call the resulting logic **channel**-CPDL^s.

The other alternative is to consider essentially **channel**-CPDL models. This requires us to restrict the syntax of **shared**-CPDL so that shared variables must be assigned values before they are tested. Then the semantic rules for the resulting language, **shared**-CPDL^c, will construct the interpretation I_s for shared variables gradually, defining $I_s(X)$ only after X was assigned to for the first time.

PROPOSITION 6.4. (1) **channel**-CPDL^s = **shared**-CPDL.

(2) **channel**-CPDL = **shared**-CPDL^c.

Proof. The ($\leqslant$) direction of both cases is just as in Lemma 6.1. For the other direction, let the equivalent of $A \in$ **shared**-CPDL be $\bigvee_\beta ((\bar{X}=\bar{b}) \wedge A_\beta)$ in **channel**-CPDL^s, and let the equivalent of $A \in$ **shared**-CPDL^c be A_β for an arbitrary β, where the formula A_β is the β-equivalent of A in **channel**-CPDL. ▮

Next we show that in expressive power

$$\text{CPDL} \leqslant \textbf{count}\text{-CPDL} < \textbf{u-count}\text{-CPDL} \leqslant \textbf{channel}\text{-CPDL}.$$

We conjecture that the first inequality is strict too. For instance, we suspect that the formula $even_2$ of **count**-PDL, given in Example 5.1, is inexpressible in CPDL. However, we can prove strictness only for the middle inequality.

The first inequality is trivial. Strictness of the second inequality is shown by complexity considerations on the satisfiability problem of the involved languages.

LEMMA 6.5. *The formula grid(P) of Example* 5.2 *(for an atomic formula P) is inexpressible in* **count**-CPDL.

Proof. Assume that there is a **count**-CPDL formula $g(P)$ equivalent to *grid*(P). Then for every formula ψ in *prop. calculus*, the formula *grid*(ψ) of *grid*-PDL can be

expressed in **count**-CPDL by replacing every occurrence of P in $g(P)$ with ψ. This means that *grid*-PDL is effectively translatable into **count**-CPDL. On the other hand, in the next section it is shown that **count**-CPDL is decidable while *grid*-PDL is not; a contradiction. ∎

COROLLARY 6.6. **count**-CPDL < **u-count**-CPDL.

Finally we show the third inequality.

PROPOSITION 6.7. **u-count**-CPDL ≤ **channel**-CPDL.

Proof. **channel**-CPDL can represent the main counter controls of a **u-count**-CPDL formula by means of additional processes, receiving the messages from all other processes and counting them. Let $A = \langle \alpha \rangle_{spec} B$ be a (singly specified) formula employing messages $M_1,\ldots, M_k$. By Lemma 5.2 we assume no specifications of type (S3) are present. Let α' be the program obtained by replacing every occurrence of M_i with $C_i!1$, where C_i is a new channel, for every $1 \leqslant i \leqslant k$. By the inductive hypothesis, let B' be a **channel**-CPDL formula equivalent to B.

For any specification statement *stat* in *spec* construct a counting process $count_{stat}$, according to the type of the message:

(S1) $\#M_i = l \pmod t$: let $count_{stat}$: $((C_i?1)^t)^*; (C_i?1)^l$.

(S2) $\#M_i = l$: let $count_{stat}$: $(C_i?1)^l$.

(S4) $\#M_i = \#M_j$: let $count_{stat}$: $(true? \cap true?)^*; (C_i?1 \cap C_j?1)$.

This last program spawns an arbitrary (even) number of parallel processes, half of which wait to read 1 in channel C_i, and the other half expect 1 in C_j.

Now A can be replaced by the **channel**-CPDL formula

$$\left\langle \alpha'; B'? \cap \bigcap_{stat \in spec} count_{stat} \right\rangle true. \quad ∎$$

7. *Decidability Issues*

Example 3.2 can be easily used to prove that the validity problem for **channel**-CPDL is undecidable, by a reduction from Post Correspondence Problem (PCP). For a given instance of PCP, we can construct a **channel**-CPDL formula $pcp_{x,y}$ which is satisfiable iff the given instance has a solution. Specifically, it is necessary to force the model to be of the desired form of a single path (or several isomorphic ones), and then state $correspond_{x,y}$. Details appear in [Pe2].

Thus the ability to synchronize two parallel processes renders the validity problem of **channel**-CPDL undecidable. Note that in this example, the ability to spawn an unbounded number of processes has not been used; in fact, at most two processes participated in any program.

We now show that the validity problems of **channel**-CPDL, **u-count**-CPDL and even *grid*-PDL are actually Π_1^1-hard. This stems from the capability of unbounded

comparisons, and is shown by reducing a certain *recurring domino problem* to the satisfiability problem of *grid*-PDL. Note that here we do not use any form of synchronization. Thus each of these two capabilities by itself is powerful enough to cause undecidability.

The proof goes along the lines of several similar proofs in [H3]; in particular, that non-inference and non-implication in PDL are Σ_1^1-hard.

The input to the domino problem is a finite set $T = \{d_0, ..., d_m\}$ of domino types, each of the form $d_i = (left_i, right_i, up_i, down_i)$, giving the four colors associated with the sides of d_i. Colors are taken from the set $C = \{c_1, ..., c_k\}$. The problem is stated as follows: Given T, can T tile the grid $N \times N$ such that d_0 occurs in the tiling infinitely often in the first column?

PROPOSITION 7.1 [H2, H3]. *This domino problem is Σ_1^1-complete.*

The colors of each square on the grid are coded by $4k$ atomic predicates $\{L_i, R_i, U_i, D_i \mid 1 \leqslant i \leqslant k\}$. For instance, the fact that the left side is coloured with c_i is expressed by the formula

$$(LEFT = c_i)\colon L_i \wedge \bigwedge_{\substack{1 \leqslant j \leqslant k \\ j \neq i}} (\neg L_j).$$

The formula

$$(TILE = d_i)\colon LEFT = left_i \wedge RIGHT = right_i \wedge UP = up_i \wedge DOWN = down_i$$

states that the square is tiled by d_i.

The formula R_T forces the model to be in the form of the grid described in Fig. 5.1 and tiled in the desired way. The necessary formulas are

inftree:	$[(a \cap b)^*](\langle a \rangle \mathit{true} \wedge \langle b \rangle \mathit{true})$,
T-tiled:	$[(a \cup b)^*](\bigvee_{0 \leqslant i \leqslant m} (TILE = d_i))$,
same-tiling:	$\bigwedge_{0 \leqslant i \leqslant m} grid(TILE = d_i)$
sides-match:	$[(a \cup b)^*](\bigwedge_{1 \leqslant i \leqslant k} ((UP = c_i \supset [b]\, DOWN = c_i)$
	$\wedge\ (RIGHT = c_i \supset [a]\, LEFT = c_i)))$,
d_0-*recurring*:	$[b^*]\langle b^* \rangle\, TILE = d_0$.

Finally

$$R_T\colon \quad \mathit{inftree} \wedge \mathit{T\text{-}tiled} \wedge \mathit{same\text{-}tiling} \wedge \mathit{sides\text{-}match} \wedge d_0\text{-}\mathit{recurring}.$$

Clearly R_T is satisfiable iff the model can be unwound into an a/b tree representing a grid (i.e., states reachable by executions of different permutations of

the same seq have similar colouring), and d_0 occurs infinitely often in the first column. Hence

PROPOSITION 7.2. *The satisfiability problem of grid-*PDL *is* Σ_1^1*-hard; the validity problem of grid-*PDL *is* Π_1^1*-hard.*

COROLLARY 7.3. *The validity problem of* **u-count**-CPDL, **channel**-CPDL, *and* **shared**-CPDL *is* Π_1^1*-hard.*

Next we show that the validity problem of **count**-CPDL is decidable by reducing it to **c-goto**-CPDL.

Let $\langle\alpha\rangle_{spec} A$ (or the equivalent $\langle\alpha; A?\rangle_{spec}\, true$) be a singly specified **count**-CPDL formula. For simplicity assume α contains a single message M and *spec* contains a single specification $\#M = j \pmod k$ (specifications of type (S2) and the cases of several specifications and several messages are handled by an appropriate generalization).

First transform α into an equivalent concurrent goto scheme with messages, $\alpha' = (1: \gamma_1, \ldots, m: \gamma_m)$. The time required for this translation and the size of the resulting program are both linear in the size of the original program.

Translating α' into **c-goto**-CPDL without messages proceeds as follows. Construct k identical copies of α, denoted $\alpha^0, \ldots, \alpha^{k-1}$, with the labels of α^i being $1^i, \ldots, m^i$. The idea is that a run of the new program will reach label l^i just when the original α had to reach label l and in the computation of the original program, the subtree rooted in the corresponding state contains $i \pmod k$ occurrences of M.

Now change the program as follows. Replace every occurrence of l^i: M by l^i: *goto* $(l+1)^{(i-1) \bmod k}$, and every occurrence of l^i: *goto* l_1^i *and* l_2^i by l^i: $\bigvee_{0 \leqslant t \leqslant k-1}$(*goto* l_1^i *and* $l_2^{(i-t) \bmod k}$) (this informal writing corresponds to splitting nondeterministically into k possible continuations, resulting in $2k-1$ lines of code).

Finally add to every copy α^i where $i \neq 0$ a line $m+1$: *false*?, and concatenate all the k schemes (consistently renaming labels) into a single scheme β, starting with the scheme α^j and finishing with α^0 (the other copies may appear in any order). The formula $\langle\beta\rangle\, true$ is then equivalent to the original. Hence we have shown

PROPOSITION 7.4. **count**-CPDL $\leqslant$ **c-goto**-CPDL.

Together with Proposition 2.1 we can now obtain an elementary procedure for validity in **count**-CPDL, where the exact complexity depends on the representation of specifications. If a specification $\#M = j \pmod k$ is written with a binary representation for j and k, then the translation causes an exponential blowup. In the general case where there are several messages (which may be multiply specified) we get an exponential blowup in the *square* of the formula size. Therefore we have

PROPOSITION 7.5. *The validity problem for* **count**-CPDL *can be decided nondeterministically in time triple exponential in* n^2*, where* n *is the size of the formula.*

PART C: THE FIRST-ORDER LEVEL

8. *The Logics*

8.1. **channel**-CQDL

The language **channel**-CQDL relates to CQDL just as **channel**-CPDL relates to CPDL, except that the messages are not necessarily Boolean. The syntax is that of CQDL (Sect. 2.4), with the addition of the channel communication operations $C!x$, meaning "transmit the value of x over channel C," and $C?y$, meaning "receive the message from channel C into y."

The semantics is based on a first-order structure $\mathscr{A} = \langle \mathscr{D}, P_1, \ldots, f_1, \ldots \rangle$, and employs a set of S of states, which are determined, as in CQDL, by the values of all involved variables. The values of the interpretation functions π and ρ are determined as in **channel**-CPDL, with the following modifications:

(a) On the first-order level, atomic formulas are predicates $P(\bar{\sigma})$ where $\bar{\sigma}$ is a tuple of terms, and are assumed to include equality. Their truth-value in a state is determined by the value of $P(\bar{\sigma}_s)$ in the structure, where $\bar{\sigma}_s$ is the evaluation of $\bar{\sigma}$ in the state s.

(b) Atomic programs here are assignments $x \leftarrow \sigma$, for a term σ. As in CQDL, we let $\rho(x \leftarrow \sigma) = \{(s, s') \mid s' = s[\bar{\sigma}_s/x]\}$, where σ_s is the evaluation of σ in s.

(c) The semantic rule regarding communication becomes

if $\alpha_i = C!x; \beta_1$, $\alpha_j = C?y; \beta_2$, $j > i$ and

$$(\bar{s}(1, j-1) \mid s_j[x/y] \mid \bar{s}(j+1, n), \bar{r}) \in \rho(\bar{\alpha}(1, i-1) \mid \beta_1 \mid \bar{\alpha}(i+1, j-1) \mid \beta_2 \mid \bar{\alpha}(j+1, n))$$

then $(\bar{s}, \bar{r}) \in \rho(\alpha)$.

A similar rule applies for the case when $j < i$.

8.2. **shared**-CQDL

The language of **shared**-CQDL has also a set of shared variables $\{X_i\}$. Here, too, the semantics is based on a first-order structure as above, and extends that of CQDL as **shared**-CPDL extends CPDL. Thus a super-state $\hat{s}$ consists of a vector $\bar{s} = [s_1 \mid \cdots \mid s_n]$ of states, with each s_i containing a separate valuation for the ordinary (private) variables, and a valuation I_s: $\{X_i\} \to \mathscr{D}$ for the shared variables.

Shared variables can be used just as any other variables in programs and formulas. Nevertheless, it is clear that on the first-order level too we must have the test-and-set command, $TS(X)$.

The appropriate semantic rules are:

Semantic rule for $X \leftarrow \sigma$,

if $\alpha_i = X \leftarrow \sigma; \beta$, $\hat{s} = (\bar{s}, I_s)$,
σ_s is the evaluation of the term σ in $\hat{s}$
and $((\bar{s}, I_s[\sigma_s/X]), \hat{r}) \in \rho(\bar{\alpha}(1, i-1) \mid \beta \mid \bar{\alpha}(i+1, n))$
then $(\hat{s}, \hat{r}) \in \rho(\alpha)$.

Semantic rule for TS(X),

$$\begin{aligned}&\text{if } \alpha_i = TS(X);\ \beta,\ \hat{s} = (\bar{s}, I_s),\ I_s(X) = 0,\\ &\text{and } ((\bar{s}, I_s[1/X]),\ \hat{r}) \in \rho(\bar{\alpha}(1, i-1)\,|\beta|\,\bar{\alpha}(i+1, n)),\\ &\text{then } (\hat{s}, \hat{r}) \in \rho(\bar{\alpha}).\end{aligned}$$

Note. Here and in several places hereafter, we implicitly assume the existence of at least two distinct elements in the structures domain, namely 0 and 1. (This is needed, for instance, for binary encoding of information in certain simulations.) In all cases it is clear that the special case of a singleton universe can be taken care of separately (i.e., by an additional disjunct in the appropriate formula).

8.3. **count**-CQDL

We let the counting mechanism into **count**-CQDL just as in the propositional level; as long as the control specification has to specify the desired messages explicitly, it gives no additional power to allow general messages (i.e., the contents of a variable). In fact, in the next section we show that this system is no more expressive than CQDL itself.

8.4. *The kill Command*

It is sometimes desirable to be able to terminate the execution of some of the processes of a concurrent program. For example, we may want to run several processes of a program α in parallel, pass some information through the channels, and then terminate some of the processes and proceed in executing a program β in the remaining processes. Recall that in the present situation, $\alpha;\beta$ is interpreted so that β is executed from the end-state of *every* branch of the executed trec of α, and similarly, $\langle\alpha\rangle A$ is interpreted so as to evaluate A in every leaf of the executed trec.

The *kill* command described in [Pe3] enables the termination of some branches of a trec, so that further subprograms, and the final evaluation of the formula, will not refer to these branches.

The *kill* command may be defined also in the two extensions described here. For instance, the semantic rule for *kill* in **channel**-CQDL is

$$\begin{aligned}&\text{if } \alpha_i = \mathit{kill};\ \beta\\ &\text{and } (\bar{s}(1, i-1)\,|\bar{s}(i+1, n),\ \bar{r}) \in \rho(\bar{\alpha}(1, i-1)|\ \bar{\alpha}(i+1, n))\\ &\text{then } (\bar{s}, \bar{r}) \in \rho(\bar{\alpha}).\end{aligned}$$

In [Pe3] it is shown that the *kill* command is actually redundant, as it is expressable in CQDL. The same holds for the extensions proposed here. The elimination of *kill* commands is carried out inductively. Given $\langle\alpha\rangle A$ (where A contains no appearances of *kill*, by the inductive hypothesis), we construct the formula

$$A'\colon\quad \langle x \leftarrow 1;\alpha'\rangle(x = 0 \vee (x = 1 \wedge A)),$$

where x is a new variable, and α' is obtained from α by replacing every subprogram β with $x = 0?\cup(x = 1?;\beta)$, and every *kill* command with $x \leftarrow 0$.

Note. The "Temporary Halt" technique. The same method enables us to suppress some branches of a trec temporarily, and then re-activate them, either by re-assigning $x \leftarrow 1$ again, or by eliminating the tests $x = 1?$ from some point on. This technique will be used extensively in the following section.

9. *Expressiveness Results*

In this section we give some expressiveness results concerning the defined logics on the first order level. Let us begin by showing that the counting mechanism adds no power to CQDL.

PROPOSITION 9.1. **count**-CQDL = CQDL.

Proof. Let $\langle\alpha\rangle_{spec} A$ be a singly specified CQDL formula. The message collecting process for a specification $\# M = l \pmod t$ can be simulated as follows. A $\lceil log\ t \rceil$ tuple of variables $\bar{z} = (z_1, \ldots, z_{\lceil log\ t \rceil})$ will hold, in any state s during the computation, the binary representation of the number of M messages (modulo t) expected to be transmitted by the sub-trec rooted at s. Initially, $\bar{z}$ holds l. Every occurrence of M is replaced by $\bar{z} \leftarrow \bar{z} - 1 \pmod t$, and every subprogram $\beta \cap \gamma$ is replaced by

$$\bigcup_{0 \leqslant i \leqslant t-1} (\bar{z} \leftarrow i; \beta \cap \bar{z} \leftarrow \bar{z} - i \pmod t; \gamma).$$

Similar treatment is given to specifications of type (S2).

The final program α' contains no occurrences of messages, but involves tuples $\bar{z}_M$ for every message M in the original α. The final formula is $\langle\alpha'\rangle(A \wedge \bigwedge \bar{z}_M = 0)$, where the conjunction is over all message types in α. ∎

Next we show that **channel**-CQDL and **shared**-CQDL are fully equivalent in expressive power.

THEOREM 9.2. **channel**-CQDL = **shared**-CQDL.

Proof. ($\leqslant$) The simulation of channels by shared variables goes just as in the propositional case, only modified to account for the general (non-Boolean) nature of the messages.

($\geqslant$) The main problems are the simulation of a shared variable X *within* a program and *outside* programs, and the simulation of $TS(X)$.

Let X be a shared variable in a formula α of **shared**-CQDL. In **channel**-CQDL we use a distinguished (private) variable x to represent X. Outside programs, x is used directly to replace X. Within a program α, in the context of a sub-formula $\langle\alpha\rangle B$, simulate X by a process $simul_x$, using two channels, G_x, C_x, and a new (private) variable y, not appearing in A. Replace $\langle\alpha\rangle B$ with

$$\langle(\alpha'; C_x?x) \cap (simul_x; kill)\rangle B,$$

where

$$simul_x: \quad (C_x?x \cup C_x!x \cup (G_x?y; x = 0?; x \leftarrow 1))^*,$$

and α' is obtained from α by the following modifications:

(1) Every assignment $X \leftarrow \sigma$ is replaced by $x \leftarrow \sigma; C_x!x$.

(2) Every occurrence of an assignment $z \leftarrow \sigma(X)$ is replaced by $C_x?x; z \leftarrow \sigma(x)$.

(3) Every occurrence of $TS(X)$ is replaced by $G_x!1$.

(4) A test $D?$ containing occurrences of X is translated internally into D', and then replaced by $C_x?x; D'?$. This is meant to ensure that the evaluation of D' starts with the current, up-to-date value of X. (Note that each program within D' has its own mechanism $simul_x$ for handling X.)

This modification has to be repeated for *every* shared variable appearing in the program α. ▮

Our next goal is to relate these systems in expressiveness to other systems of dynamic logic.

We consider the following versions. QDL_{proc} is obtained from QDL by enriching the control structure of programs with recursive procedures with parameter. The language **arr**-QDL is obtained by allowing *array assignments*, i.e., assignments of the form $f(x) \leftarrow \sigma$, where f is a function symbol and σ is a term.

Now consider a structure $\mathscr{A}$ with function symbols $f_1,..., f_l$ and constants $d_1,..., d_m$, and let $\bar{x} = (x_1,..., x_n)$ be a tuple of free variables. Define the *Herbrand universe* $HU(\bar{x})$ as the subset of the domain of $\mathscr{A}$ consisting of all elements equivalent to terms of the language over $\bar{x}$ (i.e., reachable from $d_1,..., d_m$, $x_1,..., x_n$ by means of $f_1,..., f_l$). Define $Hran(\bar{x}, z)$ as a program assigning to z an arbitrary element of $HU(\bar{x})$. Denote by **Hran**-CQDL the language of CQDL enriched with the *Hran* operation.

PROPOSITION 9.3 [Pe1]. $\text{CQDL} < \text{QDL}_{\text{proc}} = \textbf{Hran}\text{-CQDL}$.

LEMMA 9.4. *Hran* is programmable in **channel**-CQDL.

Proof. Assuming the rank of each function f_i is r_i $(1 \leqslant i \leqslant l)$, $Hran(\bar{x}, z)$ can be programmed as

$$\begin{aligned}
&((true? \cap true?)^*;\\
&(\textstyle\bigcup_{1 \leqslant i \leqslant n} C!x_i \cup \bigcup_{1 \leqslant i \leqslant m} C!d_i\\
&\cup \textstyle\bigcup_{1 \leqslant i \leqslant l} C?y_1;...; C?y_{r_i}; y \leftarrow f_i(y_1,..., y_{r_i}); C!y); kill)\\
&\cap C?z.
\end{aligned}$$

The main part of this program spawns an arbitrary number of processes, each of which is responsible for issuing an element of the Herbrand universe through chan-

nel C. Either it sends some d_i or x_i, or it accpets r_i values $y_1,..., y_{r_i}$ through C, and sends back $f_i(y_1,..., y_{r_i})$. ▮

COROLLARY 9.5. $\mathrm{QDL}_{\mathbf{proc}} \leqslant$ **channel**-CQDL.

COROLLARY 9.6. CQDL $<$ **channel**-CQDL.

As an upper bound we would like to compare **channel**-CQDL with **arr**-QDL.

Denote by (s, ∞) the collection of infinite denumerable structures $\langle \mathscr{D}, suc, d,... \rangle$ with an infinite acyclic successor function *suc* over the domain, starting with the constant d. Over such structures, **channel**-CQDL can be shown no more expressive than **arr**-QDL.

Note. Even on these "arithmetical" structures (cf. [H1]) we compare expressiveness in the strong sense of *uniform* translations: for two logics L_1, L_2, and a collection G of structures, we say that $L_1 \leqslant_G L_2$ if for every formula $A_1 \in L_1$ there is a formula $A_2 \in L_2$ s.t. $\pi(A_1) = \pi(A_2)$ in every structure of G. In contrast, the expressiveness comparisons stated in [H1] between languages over arithmetical structures are non-uniform; they supply different formulas A_2 as equivalents of A_1 over different structures.

THEOREM 9.7. **channel**-CQDL $\leqslant_{(s,\infty)}$ **arr**-QDL.

Proof. The simulation of a **channel**-CQDL program involves a complete trace of the run of the program, according to the semantics of **channel**-CQDL. Suppose we want to simulate a program α with n variables $\bar{x} = (x_1,..., x_n)$. Assume that in some given moment there are k active processes, $p_1,..., p_k$. In the simulating program, these processes are identified by $d_0, d_1,..., d_{k-1}$, where $d_i = suc^i(d)$ (in particular, $d_0 = d$). The function f is used to keep track of the current location in the program, and the functions $g_1,..., g_n$ store the values of $\bar{x}$, in each process. More precisely, $g_1(d_i),..., g_n(d_i)$ keep the current values of the variables $x_1,..., x_n$ in the process p_i, i.e., they represent the state in which the process is. As for the "next command" of each process, we apply the same method used to simulate CQDL programs by $\mathrm{QDL}_{\mathbf{proc}}$ programs in [Pe1]. Define $sub(\alpha)$ as the collection of subprograms of α, and attach a label l_β to every $\beta \in sub(\alpha)$; also define l_{end} to denote the end of the program. Now $f(d_i)$ will contain the label of the current command to be executed by process p_i.

Next, assign a set N_β of labels to every $\beta \in sub(\alpha)$. The labels of N_β are those to which one can proceeed after executing β. These sets are defined by induction, as follows:

$\beta = \alpha$ (the entire program):	let	$N_\alpha = \{l_{end}\}$,
$\beta = \gamma \cup -\delta$:	let	$N_\gamma, N_\delta = N_\beta$,
$\beta = \gamma \cap \delta$:	let	$N_\gamma, N_\delta = N_\beta$,
$\beta = \gamma; \delta$:	let	$N_\gamma = \{l_\delta\}$, $N_\delta = N_\beta$,
$\beta = \gamma^*$:	let	$N_\gamma = \{l_\beta\} \cup N_\beta$.

A formula $\langle\alpha\rangle A$ is simulated by a formula of the form $\langle init; \alpha'\rangle A'$. The variables are initialized by

$$init: \quad K \leftarrow d; f(K) \leftarrow l_\alpha; \bar{g}(K) \leftarrow \bar{x}.$$

(We abbreviate $x_1 \leftarrow g_1(y);\ldots; x_n \leftarrow g_n(y)$ by $\bar{x} \leftarrow \bar{g}(y)$, and similarly for $\bar{g}(y) \leftarrow \bar{x}$ and the like.) The variable K counts the active processes: if there are k active processes then $K = d_{k-1}$.

The main body of the simulation is based on a collection of procedures simulating the execution of subprograms in the different processes. Define a procedure $rule_\beta$ for each $\beta \in sub(\alpha)$, except for communication commands, and a procedure $rule_{\beta,\gamma}$ for each matching pair of communication commands (i.e., with the same channel and complementary punctuation). We make use of the program

$$pr\text{-}id(y)\colon y \leftarrow d;\ z \leftarrow d;\ (z \leftarrow suc(z);\ (true? \cup y \leftarrow suc(y)))^*;\ z = K?,$$

which nondeterministically assigns to y an identifier of some active process.

The *rule* procedures are defined as follows:

For $\beta = \gamma \cup \delta$,

$$rule_\beta: \quad pr\text{-}id(y); f(y) = l_\beta?;\ (f(y) \leftarrow l_\gamma \cup f(y) \leftarrow l_\delta).$$

For $\beta = \gamma; \delta$,

$$rule_\beta: \quad pr\text{-}id(y); f(y) = l_\beta?; f(y) \leftarrow l_\gamma.$$

For $\beta = \gamma^*$,

$$rule_\beta: \quad pr\text{-}id(y);\ f(y) = l_\beta?;\ (f(y) \leftarrow l_\gamma \cup \bigcup_{l \in N_\beta} f(y) \leftarrow l).$$

For $\beta = \gamma \cap \delta$,

$$rule_\beta: \quad pr\text{-}id(y); f(y) = l_\beta?; f(y) \leftarrow l_\gamma;$$
$$K \leftarrow suc(K); f(K) \leftarrow l_\delta; \bar{g}(K) \leftarrow \bar{g}(y).$$

For $\beta = x_i \leftarrow \sigma$,

$$rule_\beta: \quad pr\text{-}id(y); f(y) = l_\beta?; \bar{x} \leftarrow \bar{g}(y); g_i(y) \leftarrow \sigma; \bigcup_{l \in N_\beta} f(y) \leftarrow l.$$

For any pair $\beta = C!x_i$, $\gamma = C?x_j$,

$$rule_{\beta,\gamma}: \quad pr\text{-}id(y_1); pr\text{-}id(y_2); f(y_1) = l_\beta?; f(y_2) = l_\gamma?;$$
$$g_j(y_2) \leftarrow g_i(y_1); \bigcup_{l \in N_\beta} f(y_1) \leftarrow l; \bigcup_{l \in N_\gamma} f(y_2) \leftarrow l.$$

Each procedure *rule* mimics some semantic rule of **channel**-CQDL in some process. Thus each activation of some $rule_\beta$ corresponds to advancing the super-

state, or cut, one step down the computation tree, as described in Fig. 3.1. For instance, in the case of $\beta = x_i \leftarrow \sigma$, procedure $rule_\beta$ first picks up some process-*id* j and stores it in y. It then checks that process j indeed has to execute β in its next step, and simulates it (by assigning σ to $g_i(j)$). Finally, j's program counter $f(j)$ is adanced to the next command to be executed after β (or one of N_β, if several possibilities exist).

All the rules are now combined together into

$$\alpha': \quad \left(\bigcup rule_\beta \cup \bigcup rule_{\beta,\gamma}\right)^*.$$

Finally, in A' we have to verify that B holds in any final state of any of the active processes, where B is the **arr**-QDL equivalent of A, guaranteed by the inductive hypothesis

$$A': \quad \langle y \leftarrow d; (f(y) = l_{end}?; \bar{x} \leftarrow \bar{g}(y); B?; y \leftarrow suc(y))^* \rangle\, y = suc(K). \quad \blacksquare$$

In the other direction, it seems that a full simulation of arrays in **channel**-CQDL might not be possible. Indeed, an array f can be simulated *inside* the main flow of a program α [Pe2], relying on the fact that any specific run of an **arr**-QDL program utilizes only a finite segment of f. However, this simulation fails when f is to be additionally used (after being changed) either outside the $\langle\alpha\rangle$ connective or *within tests* in α.

Finally we consider a version with bounded degree of concurrency. Denote by **channel-bc**-CQDL the restriction of **channel**-CQDL to programs with no occurrences of $\cap$ under $*$.

PROPOSITION 9.8. **channel-bc**-CQDL = QDL.

Proof. One direction is immediate. For the other direction, use induction on the structure of a formula in **channel-bc**-CQDL. In particular, let $\langle\alpha\rangle$ *true* be a formula in **channel-bc**-CQDL, where by the inductive hypothesis every test of α is a QDL formula. Translate α into **shared**-CQDL (i.e., into a program using shared variables instead of channels). Note that by our translation procedure, the resulting program must also have a fixed number of processes; one for each *process* of the original α, and one for each *channel* appearing in α.

Now transform (the graph description of) α into a tree form by repeatedly applying the following rewrite rules to any subprogram of α, whenever possible:

$$(\beta \cup \gamma); \delta \Rightarrow \beta; \delta \cup \gamma; \delta,$$

$$(\beta \cap \gamma); \delta \Rightarrow \beta; \delta \cap \gamma; \delta.$$

Next, merge concurrent subprograms $\beta_1 \cap \beta_2$ into sequential programs, cancelling the use of $\cap$. This is done inductively, starting with the innermost $\cap$, in the following way. Let $\bar{x} = (x_1, \ldots, x_n)$, $\bar{X} = (X_1, \ldots, X_m)$ be the tuples of private and

shared variables of some subprogram $\beta_1 \cap \beta_2$, respectively. In each of the two subprograms β_i, $i=1, 2$, replace $\bar{x}$ with a tuple of new variables $\bar{x}^i = (x_1^i, ..., x_n^i)$, so that each β_i now uses a set of distinct private variables. Now, regard β_1 and β_2 as regular expressions over an alphabet containing all involved tests $\varphi?$, assignments $x \leftarrow \sigma$, $X \leftarrow \sigma$ and test-and-set commands $TS(X)$, and construct the regular expression γ corresponding to the shuffle of β_1 and β_2. Finally let $\delta = \bar{x}^1 \leftarrow \bar{x}; \bar{x}^2 \leftarrow \bar{x}; \gamma$. This results in a single sequential program equivalent to the original $\beta_1 \cap \beta_2$. Proceed in this fashion until there are no more occurrences of $\cap$ in the program. Finally replace any occurrence of $TS(X)$ with $X=0?; X \leftarrow 1$. Call the resulting program α'. Then $\langle \alpha' \rangle$ *true* is the QDL-equivalent of $\langle \alpha \rangle$ *true*. ▮

10. *Discussion*

The computation model underlying the programs of CDL is that of and/or trees, appearing in various forms in languages and systems such as *concurrent goto programs* [Ma, Ch], *Alternating Turing Machines* [CKS], *Concurrent And/Or programs* [HN], and *Concurrent Prolog* [S1]. We discuss some of the semantical considerations and choices involved in the definitions of our languages by comparing the program schemes of **channel**-CQDL to those of CSP, the concurrent programming language of Communicating Sequential Processes [Ho].

One major difference between the two systems is that in programs of CSP, the tree-like behaviour is restricted to the scope of a *parallel command* $[\alpha_1 \| \cdots \| \alpha_n]$. This means that the execution of the next command starts only after the termination of all the processes of the current parallel command. Hence, the fundamental program transformation enabling the global tree behaviour in our languages, $(\theta \| \gamma); \delta \Rightarrow (\theta; \delta) \| (\gamma; \delta)$, does not hold in CSP.

Another difference between our programs and the original version of CSP (as decribed in [Ho]) is that in CSP, the number of processes is pre-defined, hence bound by a constant. As we have seen, a restriction of **channel**-CQDL under this requirement yields a language no more expressive than regular QDL. Several papers have since augmented CSP by eliminating this restriction.

Yet another difference is that in the original CSP, communication is achieved by *process naming* rather than via channels; the processes are identified by their names, and a process P_i can send the value of x to P_j by issuing the command $P_j!x$. This message is received by P_j into y by simultaneously executing $P_i?y$. We have used Milner's channel naming instead [Mi], and our communication method is, therefore, the same as that of [HN].

However, the two methods of communication are equivalent when the number of processes is fixed. Clearly process naming can be simulated by channels, using a dedicated channel C_{ij} for all messages from P_i to P_j. Conversely, channels can be simulated by process naming in the following way. Suppose the channel C is used in a program α for sending messages from processes $P_1, ..., P_k$ and for accepting messages in $Q_1, ..., Q_l$. Then in any process R, any appearance of $C!x$ can be

replaced by $\bigcup_{1 \leq i \leq l, Q_i \neq R} Q_i!x$, and any appearance of $C?x$ can be replaced by $\bigcup_{1 \leq i \leq k, P_i \neq R} P_i?x$.

There are several other differences which may be viewed as "syntactic sugaring" of CSP. Such a difference is the existence of arrays. Obviously, pre-defined, fixed-length arrays can be simulated in regular QDL by a collection of individual variables, plus a tuple of variables serving as a pointer.

One should note that for comparisons over our uninterpreted structures, expressions of CSP must also be uninterpreted; for instance, arithmetical built-in functions such as "plus" are not allowed, unless supplied by the underlying structure.

The semantics given here for our languages is essentially of operational nature. The definitions describe the operation of a program as a whole, based on its computation trees. We would like to be able to characterize the basic program connectives by means of simple denotational rules, or "axioms," as in PDL or CPDL, and thus define the meaning of a program by induction on its structure. Several works have proposed denotational semantics for CSP and related models of parallelism, like those of [Ml, HBR]. Most such semantics (e.g., in [FHLdR, MM, FLP]) are based on defining the interpretation for each process separately, and then binding them together to give the joint meaning of a program. This calls for a more complex semantical domain, and essentially requires the semantics to record the history of communications for each process, in addition to its states. It seems plausible that such semantics can be defined for our language too, and this appraoch seems a promising line for future research.

Let us now turn to some of the logical aspects of our systems. There are several branches of the field of proof systems and logics that consider parallel programs and processes. Formal (e.g., Hoare-style) proof systems exist for CSP and related models (cf. [AFdR, ChM]). An important logical framework is *temporal logic* and its concurrent version [Pn, MaP1, MaP2]. In this system it is possible to express assertions regarding the *ongoing* behaviour of a program, i.e., to refer to the intermediate states of a computation path. However, in concurrent temporal logic the basic environment is the "shared memory" model, in which the concurrent processors constantly operate on the same data (in an interleaved fashion). Our focus is slightly different, as we view processes as essentially independent, and allow interaction and cooperation only via the designated communication mechanisms. Furthermore, in the basic versions of temporal logic, the semantical world recognizes only a single, fixed program, whose representation is based on labeled program schemes rather than on structured schemes, and discussion of parallel activities might become cumbersome for an involved program. (This situation has changed in some recent work concerning concurrent and compositional temporal logic [BKP]). In contrast, DL owes its attraction in part to the fact that programs are described in it in the structured, modular form of regular expressions, enabling convenient decomposition of formulas.

CDL, and Dynamic Logic in general, is tailored towards discussing correctness issues. This makes CDL especially suitable for treating properties of concurrent

algorithms for specific computational problems. On the other hand, an apparent deficiency of CDL is that it is not properly equipped for handling issues like safety, liveness and fairness, and therefore is less suitable for dicussing distributed *systems*, in which such issues play a vital role. The reasons for this situation are rooted in a basic property of the semantics of Dynamic Logic (DL) namely, the intepretation of a program as the collection of "start–end" pairs of states of its *successful* halting runs. This has the implication that DL has no means of referring to the intermediate states of a computation, but only to its "input–output" relation. Another implication is that DL considers only finite computations, hence it cannot handle aspects of infinite computations, such as fairness, infinite loops etc. Further, DL lacks a mechanism for distinguishing between different types of failures. All failures (e.g., unsuccessful tests, inexecutable atomic programs or deadlocks) are handled the same, by omitting the corresponding run from consideration, as it does not result in a legal start-end pair of states.

Previous papers have considered extending DL to handle infinite computations ([St], cf. [H1]). It is conceivable that one can study versions of CDL which take more of the above-mentioned issues into account. Part of the solution may be achieved along the lines of the various semantics given for CSP, i.e., by adding special "deadlock" and "failure" states to the models and changing the semantic rules accordingly, resulting in a more elaborate semantics. This line of study is left for further research.

Many other interesting questions still wait to be asked and clarified. It seems to us that the situation presented in this paper calls for a thorough study of the gap between "pure" concurrency, represented by CPDL, on the one end, and full global communication, represented by **channel**-CPDL, on the other. Various limitations on the scope of communication may lead to a hierarchy of logics, in terms of expressiveness and complexity. It may also be possible to obtain axiomatizations for systems with restricted communication capabilities. In particular, such a line of research may lead to the development of a more attractive concurrent dynamic logic, with the desirable properties of admitting a reasonably powerful communication scheme, being decidable, and having a simple complete axiom system.

ACKNOWLEDGMENT

Many discussions and much advise from David Harel have helped to bring this paper to its present form.

REFERENCES

[A] K. ABRAHAMSON, "Decidability and Expressiveness of Logics of Processes," Ph.D. thesis, Univ. of Washington, 1980,

[AFdR] K. R. APT, N. FRANCEZ, AND W. P. DE ROEVER, A Proof System for Communicating Sequential Processes, TOPLAS, Vol. 2, Assoc. Comput. Mach., New York, 1980.

[BKP] H. BARRINGER, R. KUIPER, AND A. PNUELI, Now you may compose temporal logic specification, *in* "Proc. 16th ACM Symp. on Theory of Comp.," pp. 51–63, 1984.

[Ch] A. K. CHANDRA, "The Power of Parallelism and Nondeterminism in Programming," IFIP pp. 461–465, 1974.

[CKS] A. K. CHANDRA, D. C. KOZEN, AND L. J. STOCKMEYER, Alternation, *J. Assoc. Comput. Mach.* **28** (1981), 114–133.

[ChM] R. M. CHANDY AND J. MISRA, "An Axiomatic Proof Technique for Networks of Communicating Processes," Tech. Report No. TR-98, University of Texas.

[ClM] W. F. CLOCKSIN AND C. S. MELLISH, "Programming in Prolog," Springer-Verlag, Berlin, 1981.

[FHLdR] N. FRANCEZ, C. A. R. HOARE, D. J. LEHMAN, AND W. P. DEROEVER, Semantics of nondeterminism, concurrency, and communication, *J. Comput. System Sci.* **19** (1979), 290–308.

[FL] M. J. FISCHER AND R. E. LADNER, Propositional Dynamic Logic of Regular Programs, *J. Comput. System Sci.* **18** (1979), 194–211.

[FLP] N. FRANCEZ, D. J. LEHMAN AND A. PNUELI, A linear history semantics for distributed languages, *in* "Proc. 21th IEEE Symp. on Found. of Comp. Sci.," pp. 143–151, 1980.

[H1] D. HAREL, dynamic logic, in "Handbook of Philosophical Logic II," pp. 497–604, Reidel, Holland, 1984.

[H2] D. HAREL, Effective transformations on infinite trees, with applications to high undecidability, dominoes, and fairness, *J. Assoc. Comput. Mach.* **33** (1986), 224–248.

[H3] D. HAREL, Recurring dominoes: Making the highly undecidable highly understandable, *Annals Discrete Math.* **24** (1985), 51–72.

[H4] D. HAREL, "And/Or Programs: a New Approach to Structured Programming," Trans. Prog. Lang. and Systems, Vol. 2, pp. 1–17, Assoc. Comput. Mach. New York, 1980.

[Ho] C. A. R. HOARE, Communicating sequential processes, *Comm. ACM* **21** (1978), 666–677.

[HBR] C. A. R. HOARE, S. D. BROOKES AND A. W. ROSCOE, "A Theory of Communicating Processes," Tech. Report No. PRG-16, Oxford University, 1981.

[HK] D. HAREL AND D. C. KOZEN, A programming language for the inductive sets, and applications, *Inform. and Control* **63** (1985), 118–139.

[HN] D. HAREL AND S. NEHAB, "Concurrent And/Or Programs: Recursion with Communication," No. CS82-09, The Weizmann Institute of Science, Rehobot, Israel, revised version *Sci. Comput. Programming*, in press.

[HPS] D. HAREL, A. PNUELI, AND J. STAVI, Propositional dynamic logic of nonregular programs, *J. Comput. System Sci.* **26** (1983), 222–243.

[K] R. KOWALSKI, "Logic for Problem Solving," The Computer Science Library, Artificial Intelligence Series, North-Holland, Amsterdam, 1983.

[Ma] Z. MANNA, The correctness of nondeterministic programs, *Artificial Intelligence* **1** (1970), 1–26.

[Mi] R. MILNER, "A Calculus of Communicating Systems," Lect. Notes in Comput. Si., Vol. 92, Springer-Verlag, Berlin, 1980.

[MaP1] Z. MANNA AND A. PNUELI, Verification of concurrent programs: The temporal framework, *in* "The Correctness Problem in Computer Science" (R. S. Boyer and J. S. Moore, Eds.), pp. 215–273, International Lecture Series in Computer Science, Academic Press, London, 1982.

[MaP2] Z. MANNA AND A. PNUELI, Verification of concurrent programs: Temporal proof principles, *in* "Logics of Programs," (D. Kozen, Ed.), Lect. Notes in Comput. Sci., Vol. 131, pp. 200–252, Springer-Verlag, Berlin, 1982.

[MM] G. MILNE AND R. MILNER, Concurrent processes and their syntax, *J. Assoc. Comput. Mach.* **26** (1979), 302–321.

[MeP] A. R. MEYER AND R. PARIKH, Definability in dynamic logic, *J. Comput. System Sci.* **23** (1981), 279–298.

[Pe1] D. PELEG, Concurrent Dynamic Logic, *J. Assoc. Comput. Mach.* (1987), in press.

[Pe2] D. PELEG, "Communication in Concurrent Dynamic Logic," No. CS84-15, The Weizmann Institute of Science, July, 1984.

[Pe3] D. PELEG, "Concurrent Program Schemes and their Logics," CS84-25, The Weizmann Institute of Science, November, 1984.

[Pn] A. PNUELI, The temporal logic of programs, *in* "Proc. 18th IEEE Symp. on Found. of Comp. Sci.," pp. 46–57, 1977.

[S1] E. Y. SHAPIRO, "A Subset of Concurrent Prolog and Its Interpreter," No. CS83-06, The Weizmann Institute of Science, Rehovot, Israel.

[S2] E. Y. SHAPIRO, Alternation and the computational complexity of logic programs, *J. Logic Programming* **1** (1984).

[St] R. S. STREETT, Propositional logic of looping and converse is elementarily decidable, *Inform. and Control* **54** (1982), 121–141.

[T] M. TIOMKIN, "Extensions of Propostional Dynamic Logic," Ph.D. thesis, the Technion, Haifa, 1983.